Assignment 3 – Software Analysis

Extended Java Typechecking

Claudio Maggioni

1 Project selection

The assignment description requires to find a project with more than 1000 lines of code making significant use of arrays or strings.

Given these requirements, I decide to analyze the Apache Commons Text project in the GitHub repository **apache/commons-text**.

1.1 The Apache Commons Text Project

The Apache Commons family of libraries is an Apache Software Foundation[1] sponsored collection of Java libraries designed to complement the standard libraries of Java. The Apache Commons Text project focuses on text manipulation, encoding and decoding of *String*s and *CharSequence*-implementing classes in general.

All the source and test classes are contained within in the package *org.apache.commons.text* or in a sub-package of that package. For the sake of brevity, this prefix is omitted from now on when mentioning file paths and classes in the project.

I choose to analyze the project at the *git* commit `78fac0f157f74feb804140613e4ffec449070990` as it is the latest commit on the *master* branch at the time of writing.

To verify that the project satisfies the 1000 lines of code requirement, I run the *cloc* tool. Results are shown in table 1. Given the project has more than 29,000 lines of Java code, this requirement is satisfied.

Language	Files	Blank	Comment	Code
Java	194	5642	18704	26589
XML	16	205	425	1370
Text	6	194	0	667
Maven	1	23	24	536
YAML	6	39	110	160
Markdown	4	40	106	109
Velocity Template Language	1	21	31	87
CSV	1	0	0	5
Properties	2	2	28	5
Bourne Shell	1	0	2	2
Total	232	6166	19430	29530

Table 1: Output of the *cloc* tool for the Apache Commons Text project at tag *78fac0f1* (before refactoring is carried out).

[1] `https://apache.org/`

2 Running the CheckerFramework Type Checker

The relevant source code to analyze has been copied to the directory *sources* in the assignment repository

usi-si-teaching/msde/2022-2023/software-analysis/maggioni/assignment-3

on *gitlab.com*. The Maven build specification for the project has been modified to run the CheckerFramework extended type checker (version 3.33.0) as an annotation processor to be ran on top of the Java compiler. Both source code and test code is checked with the tool for violations, which are reported with compilation warnings. To run the type checker simply run:

```
mvn clean compile
```

in a suitable environment (i.e. with JDK 1.8 or greater and Maven installed). To additionally run the Apache Commons Text test suite and enable `assert` assertions (later useful for CheckerFramework `@AssumeAssertion(index)` assertions) simply run:

```
env MAVEN_OPTS="-ea" mvn clean test
```

Apache Commons Text includes classes that have been deprecated. As changing the interface and behaviour of these classes would be useless, as alternatives to them exist in the library already, I choose to ignore them while refactoring by adding a *@SuppressWarning* annotation in each of them. The state of the assignment repository after the deprecated classes are annotated and when the type checker was first ran successfully is pinned by the *git* tag *before-refactor*. A copy of the CheckerFramework relevant portion of the compilation output at that tag is stored in the file *before-refactor.txt*.

No CheckerFramework checkers other than the index checker is used in this analysis as the code in the project mainly manipulates strings and arrays and a significant number of warnings are generated even by using this checker only.

3 Refactoring

Warning type	Before refactoring	After refactoring
argument	254	241
array.access.unsafe.high	130	117
array.access.unsafe.high.constant	31	28
array.access.unsafe.high.range	22	22
array.access.unsafe.low	59	58
array.length.negative	3	3
cast.unsafe	2	2
override.return	12	12
Total	513	483

Table 2: Number of CheckerFramework Type Checker warnings by category before and after refactoring, ignoring deprecated classes.

Table 2 provides a summary on the extent of the refactoring performed in response to index checker warnings across the Apache Commons Text project. In total, 513 warnings are found before refactoring, with 30 of them later being extinguished by introducing annotations and assertions in the code in the following classes:

- AlphabetConverter
- StringSubstitutor
- similarity.LongestCommonSubsequence
- translate.AggregateTranslator
- translate.CharSequenceTranslator
- translate.CodePointTranslator
- translate.CsvTranslators
- translate.JavaUnicodeEscaper
- translate.SinglePassTranslator
- translate.UnicodeEscaper

The strategy I adopt to perform the refactoring is based on the compiler errors thrown on the original code. In every flagged statement I attempt to find the root cause of the warning and eliminate it with either extended type qualifier annotations or assertions when adding only the former fails.

Instead of using `@SuppressWarning` annotations I choose to use `@AssumeAssertion`-annotated assertions as I aim to use the existing Commons Text test suite to aid in finding incorrectly-placed annotations. As mentioned before in the report, I run the test suite of the project by enabling assertions and I verify that all tests still pass and no *AssertionErrors* are thrown.

In total, the refactor consists in the placement of 16 extended type qualifiers and 14 assertions. A more detailed description of salient refactoring decisions taken to extinguish the warnings follows.

3.1 Class *AlphabetConverter*

```
387 for (int j = 0; j < encoded.length();) {
388     final int i = encoded.codePointAt(j);
389     final String s = codePointToString(i);
390
391     if (s.equals(originalToEncoded.get(i))) {
392         result.append(s);
393         j++; // because we do not encode in Unicode extended the
394              // length of each encoded char is 1
395     } else {
396         if (j + encodedLetterLength > encoded.length()) {
397             throw new UnsupportedEncodingException("Unexpected end "
398                     + "of string while decoding " + encoded);
399         }
400         final String nextGroup = encoded.substring(j,
401                 j + encodedLetterLength);
```

Here the `substring(...)` call at line 151 is flagged by CheckerFramework warning the start and end index may be negative and that the start index may be greater than the end index. As the attribute `encodedLetterLength` is positive according to the contract of the class constructor and `j` is only incremented in the for loop or by a factor of `encodedLetterLength`, the code is correct. After introducing a `@Positive` annotation on the declaration of `j` and an `assert encodedLength > 0` after line 395, CheckerFramework agrees with my judgement.

3.2 Class *StringSubstitutor*

```
910 /** [...]
911  * @throws IllegalArgumentException if variable is not found when its allowed to
    ↪  throw exception
912  * @throws StringIndexOutOfBoundsException if {@code offset} is not in the
913  *  range {@code 0 <= offset <= source.length()}
914  * @throws StringIndexOutOfBoundsException if {@code length < 0}
```

```
 * @throws StringIndexOutOfBoundsException if {@code offset + length >
 ↪  source.length()}
 */
public String replace(final String source, final int offset, final int length) {
    if (source == null) {
        return null;
    }
    final TextStringBuilder buf = new TextStringBuilder(length).append(source,
    ↪  offset, length);
    if (!substitute(buf, 0, length)) {
        return source.substring(offset, offset + length);
    }
    return buf.toString();
}
```

The implementation of method `replace` is flagged by the extended type checker as the indices `offset` and `length` are not bound checked against the string `source`. As the unsafe behaviour of the method is documented in its *javadoc* with appropriate `@throws` clauses, I simply add this implied preconditions to the method's contract by using extended type qualifiers:

```
public String replace(final String source,
                      final @IndexOrHigh("#1") int offset,
                      final @NonNegative @LTLengthOf(value = "#1", offset = "#2 -
                       ↪  1") int length)
```

3.3 Class *translate.CharSequenceTranslator* and implementors

Apache Commons Text provides the aforementioned abstract class implementation as a template method of sorts for expressing text encoding and decoding algorithms. The class essentially provides facilities to scan UTF-16 code points sequentially, and delegating the translation of each code point to the implementation of the abstract method:

```
public abstract int translate(CharSequence input, int index, Writer writer) throws
↪  IOException;
```

CheckerFramework gives some warnings about some of the implementations of this method, highlighting that they assume the `input` *CharSequence* is non-empty and the `index` parameter is a valid index for the string.

Even if the method is public, I choose to interpret this hierarchy to mainly be a template method pattern, with high coupling between the algorithm in the abstract class and each abstract method implementation. Given this, I decide to restrict the method's precondition to highlight conditions already provided by the caller algorithm, namely the length and index constraints provided by CheckerFramework.

The new signature of the abstract method is this:

```
public abstract int translate(@MinLen(1) CharSequence input,
                              @NonNegative @LTLengthOf("#1") int index,
                              Writer writer) throws IOException;
```

As some methods have a more forgiving implementation, and a broader child method argument type from a more restrictive parent type does not break the rules of inheritance (thanks to contravariance), I choose to propagate the extended type annotations only when needed and avoid introducing additional preconditions to more tolerant implementations of the template method.

3.4 Class *translate.SinglePassTranslator* and implementors

SinglePassTranslator is one of the implementor classes of the aforementioned *CharSequenceTranslator* template method. However, the class is itself a template method pattern "for processing whole input in single pass"[2], i.e. essentially performing an abstraction inversion of the codepoint-by-codepoint algorithm in *CharSequenceTranslator* by implementing the encoding or decoding process in a single go.

The class immediately delegates the implementation of the translation algorithm to the abstract package-private method:

```java
abstract void translateWhole(CharSequence input, Writer writer) throws IOException;
```

and requires callers of the public implementation of `translate` to call it with `index` equal to 0.

I simply propagate the non-empty extended type annotation on `input` (i.e. `@MinLen(1)`) on this new abstract method and implementors.

The *translate.CsvTranslators$CsvUnescaper* implementation of this new template method requires additional attention to extinguish all CheckerFramework's index checker warnings.

```java
void translateWhole(final @MinLen(1) CharSequence input, final Writer writer) throws
↪ IOException {
    // is input not quoted?
    if (input.charAt(0) != CSV_QUOTE || input.charAt(input.length() - 1) !=
    ↪ CSV_QUOTE) {
        writer.write(input.toString());
        return;
    }

    // strip quotes
    final String quoteless = input.subSequence(1, input.length() - 1).toString();
```

Here CheckerFramework was unable to deduce that the `input.length() - 1` indeed results in a safe index for `input` as the *CharSequence* is always non-empty (as specified with the propagated type qualifiers from the abstract signature of `translateWhole`). This warning is fixed by precomputing the last index of the string and introducing a trivially true assertion on it:

```java
void translateWhole(final @MinLen(1) CharSequence input, final Writer writer) throws
↪ IOException {
    final int lastIndex = input.length() - 1;

    assert lastIndex >= 0 : "@AssumeAssertion(index): input.length() is >= 1 by
    ↪ contract";

    // is input not quoted?
    if (input.charAt(0) != CSV_QUOTE || input.charAt(lastIndex) != CSV_QUOTE) {
        writer.write(input.toString());
        return;
    }

    // strip quotes
    final String quoteless = input.subSequence(1, lastIndex).toString();
```

This assertion is one good example of CheckerFramework's limitations in verifying transitively true properties due to the lack of deductive verification capabilities. All assertions added in the project are

[2]According to the class *javadoc* description.

trivial in this way to some degree.

4 Conclusions

As evidenced by the Apache Common Text test suite and the previous section of this report, no changes in the implementation behaviour were introduced in the code by the refactor. Only extended type annotations and assertions (that hold when executing the test suite) were added to the code.

No implementaton-derived bugs were discovered during refactoring, altough the introduction of additional method preconditions via extended type annotations was sometimes needed. Some questionable design choices were found in the library, namely the *CharSequenceTranslator* template method hierarchy with `public` implementors and the inversion of abstration by refused bequest in the *SinglePassTranslator* partial implementation, however I managed to introduce correct extended types without drastic alterations (which may have broken clients of the library).

It was quite easy to introduce the CheckerFramework index checker to the Maven build toolchain of the project. However, compiling the project after this was significantly slower than before and lots of spurious warnings (i.e. false positives) were produced by the tool due to its lack of deductive verification capabilities. Some benefits were gained by using the tool, namely being able to better refine the precondition of the methods in the library by documenting them in an automatedly parsable fashion. However, the sheer number of trivially true assertions introduced to silence some warnings is a significant downside to consider when using the index checker.

The CheckerFramework extended type checker has a cost and complexity of usage greater than the Infer static analysis tool and lower than the Dafny deductive verifier. However, with respect to my experience with all the tools in carrying out the assignment, this is the tool that has shown to be the less useful.

The less powerful Infer static analyzer, even with a great number of false positives, was able to highlight some bugs in the project I have chosen for that assignment (Apache Commons Lang). As all Apache Commons libraries are quite mature and well-maintained it is expected that both tools find relatively few things to flag in them. However, no noteworthy errors were found by CheckerFramework in this project other than some small imprecisions in the precondition specification of some methods.

When compared to Dafny and deductive verifiers in general, clearly both Infer and CheckerFramework pale in comparison in terms of capabilities. However, the increased cost of adoption and of the tool's execution make it suitable only to critical portions of project codebases.

Finally, I would like to mention the Jetbrains Java Annotations as a much less powerful but effective extended type checking mechanism to handle nullable an non-null values in plain Java. Even if its scope of analysis is rather narrow and warnings an the type checking process is proprietary (as only Jetbrains IDEs interpret the annotations) I have found it an effective development aid in my university and work experience.